65C816 STRING MANIPULATION LIBRARY V3

65C816 STRING MANIPULATION LIBRARY V3 is a collection of 65C816 assembly language subroutines (aka "functions") that perform a variety of useful character string operations. The library source code may be assembled to run as a stand-alone resource (resident in any bank), or the source code may be included as part of a larger program.

This library is designed to run on any 65C816-based computer system in which the microprocessor is operated in native mode—emulation mode is not supported. The character strings that the library is able to manipulate may be located anywhere in memory and may be very large, much larger than commonly implemented in, for example, microcomputer versions of BASIC.

The library strives to be as transparent as possible to the programs that call its functions. Library functions require no pre-defined storage—any needed temporary workspace will be allocated from the hardware stack. Parameters that are required by functions are passed via the stack. The library is completely self-contained, and will be unaware of the local operating environment, assuming the latter is correctly structured—discussion on that will follow.

In writing this document, it is assumed you are reasonably conversant with the 65C816 assembly language, and know how to assemble and link programs from source code. If this is not the case, then you will need to seek help in utilizing this library—a "how-to" on assembly language development is well beyond the scope of this document.

Please carefully read this document in its entirety before attempting to use any of this software. While building and integrating the library is not difficult, some things may not be immediately obvious. If you have questions that have not been answered by this documentation please visit and post at **http://forum.6502.org**. Signing up for an account is free and takes but a few minutes.

This library is copyrighted software. Please see page 34 for terms and conditions.

GENERAL INFORMATION

Font changes are used to highlight specific aspects of the text. The main text is CG Times Roman. Alternate fonts are:

- Section headings: **Helvetica bold**.
- Subsection headings: Helvetica.
- Code examples: `Monospace`.
- Code elements in text: `Monospace`, for example, `S1`.
- Important points: *CG Times italic*, **CG Times bold** or ***CG Times bold italic***.

References to 65C816 registers are:

SYMBOL	REGISTER	SIZE IN BITS
.A	primary accumulator	8
.B	secondary accumulator	8
.C	accumulator[1]	16[1]
.X	X-index	8 or 16[2]
.Y	Y-index	8 or 16[2]
DB	Data bank	8
DP	Direct page pointer	16
PB	Program bank	8
PC	Program counter	16
SP	Stack pointer	16
SR	Status register	8
m	Accumulator/memory size	1
x	Index register size	1

---

[1] When `m=0`.

[2] 16 bits when `x=0`.

Note that `m` and `x` represent the two bits in `SR` that determine the accumulator and index register sizes.

The following abbreviations will be used to refer to some data elements:

LSB	Least-significant byte, used in reference to a 16-bit quantity.
LSW	Least-significant word, used in reference to a 32-bit quantity, e.g., a pointer to an object.
MSB	Most-significant byte, used in reference to a 16-bit quantity.
MSW	Most-significant word, used in reference to a 32-bit quantity.

LIBRARY FUNCTIONS and DISTRIBUTION FILES

The following functions are implemented in this library:

• `strcat`	Concatenate two strings.
• `strchr`	Find a byte in a string.
• `strcmp`	Compare strings.
• `strcpy`	Copy a string to a string.
• `strdel`	Delete a substring from a string.
• `strins`	Insert a substring into a string.
• `strlen`	Return the size of a string.
• `strpad`	Copy, justify and pad a string.
• `strpat`	Pattern-match strings using wildcards.
• `strstr`	Find a substring in a string.
• `strsub`	Extract a substring from a string.
• `strsws`	Strip leading and/or trailing whitespace from a string.

This distribution contains the following files:

- **`libstr30c.asm`**

  `libstr30c.asm` contains the source code that implements the various functions in the library. The source code was developed in Michal Kowalski's 6502 editor/assembler, as enhanced by Daryl Rictor to support the 65C816 assembly language. The assembler version should be at least 1.3.4 to avoid compatibility problems. The enhanced Kowalski assembler may be downloaded from Daryl's website at http://sbc.rictor.org/kowalski.html. Adapting the source code to other 65C816 assemblers will vary in difficulty. The target assembler must be able to support local labels and symbols, as well as assembly variables.

- **`stringmacs.asm`**

  `stringmacs.asm` contains macros that may be used in your programs to invoke the various functions supported by `libstr30.asm` using a higher-level syntax than what is required to directly call the library functions in "raw" assembly language. These macros use a syntax that resembles that of string manipulation functions found in a typical ANSI C library, along with means to tell the macro from where to get pointer data. Comments in `libstr30.asm` explain the details.

`stringmacs.asm` also defines some values that may be used in macro invocations or their assembly language equivalents, thus discouraging the embedding of “magic numbers” into your code. These values will be mentioned when appropriate.

Use of `stringmacs.com` is optional but recommended in programs that make a lot of library calls.

- **`libstrjt.asm`**

  `libstrjt.asm` contains a set of equates that point to the vector table in `libstr30.asm` after it has been assembled. Including `libstrjt.asm` with your programs is recommended if the library has been assembled as a relocatable standalone binary.

- **`README.pdf`**

  This document. Please advise us if you find any errors.

PROGRAMMING INFORMATION

The following general information will assist you in integrating this library in your programs.

- **This library processes ANSI C-style character strings.** Such strings consist of a contiguous series of non-null bytes followed by a null (`$00`) byte. For example, the string `ABCDE` will appear in a hexadecimal memory dump as:

  ```
  41 42 43 44 45 00
  ```

  `ABCDE` is referred to as the "text field," even if it contains one or more non-printable bytes. A string's storage size is the text field size plus one byte. Where "size" is used in reference to a string it will refer only to the text field size. A string can be null, in which case only the null terminator will be present.

- **The maximum allowable string size is 32,767 (`$7FFF`) bytes.** This size limit does not include the null terminator. Functions will terminate with an error if this limit is violated. A size error will also occur if the combined length of two strings submitted to functions such as `strcat` and `strins` exceeds the 32,767 byte limit.

- **All library functions are structured as subroutines.** During assembly of the library, conditional code makes it possible to assemble the library so "near" (`JSR`-`RTS`) or "far" (`JSL`-`RTL`) subroutine calls may be made. Near calls are practical if the library and the programs that use it are loaded into the same bank.

  Near calls will offer a small performance advantage when compared to the use of far calls. That said, assembling the source code to use far calls will permit loading the library into any bank at almost any address and calling library functions from any other bank.

- **Library object code is fully relocatable.** The library may be assembled with a starting address of `$000000` and subsequently loaded almost anywhere without any need to adjust branch and jump targets, or static references. Use of the vector table at the beginning of the `libstr30c.asm` source file means your program will only need to know the 24-bit load address of the library binary to call its functions (see also `libstrjt.asm`).

  NOTE: Library object code cannot span banks, a restriction imposed by the 65C816's architecture, not by the library itself.

- **Function call parameters are passed via a stack frame.** Each library function has specific stack frame requirements, which will be described in detail. Before control is returned to the calling program, the function will perform stack housekeeping, in which temporary workspace and the parameter stack frame will be discarded.

In other words, each library function will "rebalance the stack" upon exit and the calling program will not need to concern itself with stack housekeeping after calling a function.

WARNING! An improperly structured stack frame resulting from pushing an incorrect number of parameters or parameters of a wrong size will result in an unbalanced stack when a function terminates, likely causing system fatality. This class of error is avoidable by using the string macros to call functions.

- Function parameters consist of pointers and values. A pointer is the address of an "object" that is to be manipulated by or used by the function being called, an object being a string, numeric quantity, etc. A "value" is a datum needed by the function in order to determine how to process an object.

  ○ **A pointer is passed as a little-endian, double word**—a `DWORD`, which is a 32-bit quantity. As the 65C816 cannot natively handle a `DWORD`, it is necessary to split a `DWORD` pointer into two `WORD`s (16-bit, little-endian quantities) and sequentially push them to the stack. A method of doing so in the Kowalski assembler is:

  ```
  PEA #ADDR >> 16      ;pointer to ADDR MSW
  PEA #ADDR & $FFFF    ;pointer to ADDR LSW
  ```

  In the above, `ADDR` is the address of an object that will be known at assembly time. The expression **`>> 16`** is notation telling the assembler to right-shift `ADDR`'s value 16 bits, hence assembling the instruction with bits 16-31 of `ADDR` as the operand. The expression **`& $FFFF`** is telling the assembler to mask bits 16-31 of `ADDR`, hence assembling the instruction with bits 0-15 of `ADDR` as the operand.

  Note the order in which the pointer `WORD`s are pushed: it is always MSW first, LSW second. *Reversing the word order will create an insidious bug.* The `PEA` instruction pushes the constituent bytes of a `WORD` in the proper order.

  Unless used in self-modifying code, the `PEA` instruction can only push static data, which, in practice, limits its utility to cases in which items to be pushed are known at assembly time. If an address will be computed as the program is running and can be subsequently stored on direct page, you can use the `PEI` instruction to push it—this case would also apply to values already located on direct page.

  For example, if a computed address has been stored at `DPPTR`, `DPPTR` being the start of four contiguous bytes on direct page, the following code would push the address to the stack in the correct order, thus generating the required 32-bit, little-endian pointer:

```
PEI DPPTR+2          ;push MSW
PEI DPPTR            ;push LSW
```

The above is an example of "far" addressing, which would be used if the object is not in the same bank as the string library. Again, note the order: MSW followed by LSW. Also, note that `DPPTR+3`, which is the MSB of the MSW, must always contain `$00`—no address in a 65C816 system will ever be larger than `$00FFFFFF`. *Failure to observe this requirement may trigger internal errors in functions that carry out pointer arithmetic as they execute.*

If an object is in the same bank as that in which the library is running, it is possible to use the `PER` instruction to push the object's address, a capability that may be useful when a program's storage references could change due to a relocating load. We refer to this method as "near" addressing. The syntax to do so would be:

```
PEA #<bank>          ;push dummy MSW
PER ADDR             ;push object address LSW
```

In the above, `<bank>` would be the bank that is common to both the library and the object to be processed. As `PEA` always pushes a `WORD`, the MSB of `<bank>` should be `$00`.

An alternative to hard-coding the bank into an instruction is to determine the program's execution bank at run time and store it on direct page as a `WORD` for later use in making "near" calls. For example:

```
REP #%00100000       ;16-bit accumulator
PHK                  ;push program bank...
PHK                  ;twice
PLA                  ;load into .C
AND #%11111111       ;mask .B
STA PRGBANK          ;save bank on direct page
```

Later on, to call a function, code such as the following would be used to pass a "near" object's address:

```
PEI PRGBANK          ;push execution bank
PEA ADDR             ;push object address LSW
```

An alternative to one of the above stack-priming methods is to load the 65C816's registers with the appropriate parameters and push them. For example, the following code fragment (in the Kowalski assembler) uses `.X` and `.Y` to prime the stack with the address of `ADDR`:

```
REP #%00010000       ;16-bit index registers
LDX !#ADDR & $FFFF   ;address LSW into .X
LDY !#ADDR >> 16     ;address MSW into .Y
PHY                  ;push MSW
PHX                  ;push LSW
```

In the Kowalski assembler, the **!#** addressing mode will result in a 16-bit, immediate-mode operand being generated for the two load instructions. Use of **#** alone will result in an 8-bit operand being assembled if an operand expression resolves to less than `$0100`. Incidentally, the above sequence is an analog of the earlier example of using `PEA` to prime the stack—but less efficient.

If a computed address is not on direct page then it isn't possible to use `PEI`. In such a case, it will be necessary to load the registers from the location containing the computed address and push them, e.g.:

```
REP #%00010000       ;16-bit index registers
LDX ADDR_LOC         ;computed address LSW into .X
LDY ADDR_LOC+2       ;computed address MSW into .Y
PHY                  ;push MSW
PHX                  ;push LSW
```

In the above example, it is assumed the computed address is stored at `ADDR_LOC` in little-endian order, with `ADDR_LOC+3` containing `$00`. At the risk of being annoyingly repetitious, again note the order in which items are pushed.

Regardless of the method chosen, the goal is to generate a stack frame that is suitable for the function being called. Use the method that is most efficient and practical in your program, but avoid making it complicated. If you do have to engage in complexity be sure to thoroughly comment your code.

- **A value is passed as a WORD**. This rule applies even if the value is a character. As with pushing a pointer, you may use `PEA` to push a `WORD` whose quantity will be known at assembly time, `PEI` to push a `WORD` stored on direct page, or a register to fetch and push a `WORD` that has been stored in absolute memory. If a value is to be pushed one byte at a time, be sure to push the MSB first, followed by the LSB. *Reversing the byte order will create an insidious bug.*

In function descriptions, the difference between a pointer and a value should be clear from syntax. For example, the call to the `strpad` function take both pointers and values. The form of the call (using a macro) is:

```
strpad *S1,*S2,*L,J,FB
```

S1, S2 and L are pointers—so-indicated by the * prefixing the variable name (do not, however, use * with the pointer name in the macro invocation), whereas J and FB are values.

Lastly, when a description says a parameter is an integer it means the parameter is a 16-bit, unsigned, little-endian quantity, that is, a WORD. For example, the value 15 passed as an integer would be seen in memory as 0F 00.

- Some functions will call for an index into a string. An index is always zero-based and is always a word.

- Functions strive to be as transparent as possible. Unless a function returns data to the caller, the accumulator's and index registers' content and size will be preserved. Bits in SR other than carry will be preserved as well, except in a few cases. Registers that are guaranteed to be preserved are DB, DP, PB and SP. Refer to each function's description for more information.

- All function calls must be tested for an error upon return. If an error occurs, the function will return with the carry bit set in SR. If a function that would normally return data in one or more registers exits with an error, such registers will be unchanged from when the function was called. Each function's description will list the possible causes of an error.

- Some functions will expand S1 (string #1) during processing. Such expansion could lead to a buffer overflow error if S1 has not been allocated sufficient memory. *Library functions assume S1's buffer is of adequate size—it is the calling function's responsibility to allocate and manage storage.*

- The local operating system's interrupt service routines (ISR) must fully preserve the 65C816's state. All library functions allocate workspace on the stack and temporarily relocate direct page to that workspace as they run. Also, all functions use the MVP block-copy instruction to perform stack housekeeping before exit. ***It is imperative that ISRs fully preserve the microprocessor state and, if necessary, change DP to point to the ISR's notion of where direct page is located.***

WARNING! **Sloppy ISR design will likely cause system fatality if a library function is interrupted.** ***Your ISR should make NO ASSUMPTIONS about the microprocessor's state at the time of an interrupt's occurrence.***

If you are interested in learning more about interrupt service routine design for 65C816 systems you can find reference material at **http://sbc.steggy.net/65c816interrupts.html**.

ASSEMBLING THE LIBRARY

The following procedure describes how to assemble the library as a standalone utility using the Kowalski assembler. If you are using some other assembler your procedure will likely vary in some ways.

1. Load the `libstr30c.asm` source file into the Kowalski editor. Other than the edits described below, avoid changing anything in the source code. We recommend you make a backup copy of `libstr30c.asm` before proceeding.

2. Uncomment the following line of code:

   ```
   .org $000000
   ```

   If necessary, change `$000000` to the desired assembly address. As the binary resulting from assembly will be full relocatable, the `$000000` address may be left as is and instead, a relocating load may be performed on the target system if such a capability exists.

3. If you plan to be able to load the `libstr30c` binary to an arbitrary bank in your system, the source code must be assembled with “far” calls enabled. In such a case, uncomment the following line of code, which is immediately below the `.org $000000` line described in step 2 above:

   ```
   ;_STRLIB_   ;uncomment for "far" calls
   ```

   Do not edit the line, other than to uncomment it. With “far” calls enabled, all calls to the library must be with `JSL`, not `JSR`.

4. Strike **[F7]** to assemble the source code.

5. Strike **[Ctrl-K]** to save the object code to a file. When prompted, select the desired file type prior to saving. For transporting object code to your target 65C816 system via a TIA-232 serial link, it is suggested you select Motorola S-record or Intel hex format. If you wish to save the object code as a binary file, click the [Options] button in the Save dialog and set the proper address range. By default, the Kowalski assembler will save the entire 64 KB address space if a range is not set.

6. Transport your object code to your target 65C816 system. If you have assembled `libstr30c` to address `$000000` it will be necessary to perform a relocating load on your target system. If your system includes mass storage, save the loaded object code to a file. The library is ready for use.

7. In any program that will call library functions, you will need to include the `libstrjt.asm` source file so as to define the `libstr30c` vector table. `libstrjt.asm` contains a series of equate statements, the first of which is:

   ```
   libstr30 =$000000       ;library vector table base
   ```

   Edit the above statement so the `$000000` address conforms to the address to which the `libstr30c` binary will be loaded at run time. **DO NOT** edit anything else in the file.

   NOTE: If the `libstr30c` binary's load address will not be known at assembly time, you will have to devise some other means of informing your programs of the load address. In such case, leave the base address set to `$000000`.

8. If your program will be using the string macro definitions in `stringmacs.asm`, include that file early in the assembly chain for your program. Macros must be defined before any statements that use the macros can be assembled.

   Macros will need to know if library functions are to be accessed with near or far calls. See step 3, above.

The following procedure describes how to make the library an integral part your finished software. If you are using some other assembler, your procedure will likely vary in some ways.

1. If your program will be using the macro definitions in `stringmacs.asm`, include that file early in the assembly chain. Macros must be defined before any statements that use the macros can be assembled.

2. Include `libstr30c.asm` at any convenient point in your source file. *Do not edit anything in* `libstr30c.asm`. As the library will be part of your program once assembly is complete, *do not enable far calls*, as was described in step 3 in the previous section. All library functions will be called with `JSR`.

   Some experimentation may be required to determine at what point in your source code `libstr30c.asm` is to be included to avoid possible forward reference areas. Generally speaking, adding libraries after the code segments that call them is a good practice.

3. Add code as necessary in your program where library calls are to be made. Use of macros is recommended where possible.

FUNCTION DESCRIPTIONS

In the following text, each function description will start with the macro call syntax for the function—macro and function names are lower case, explain what the function does, describe any special processing cases that must be considered, and then finish with the assembly language call syntax (“synopsis”) applicable to the function, followed by a list of register returns. Examples will use `JSR` to call functions. If you are assembling with “far” calls enabled you should substitute `JSL`.

Strings are referred to as `S` in the descriptions for functions that work with a single string, or `S1` and `S2` in functions that work with two strings, excepting `strpat`, which refers to the strings as `S` and `P`. Other parameter references will be explained as needed.

In narrative, typical notation regarding function parameters will be as follows:

`*S1`	A *pointer* to object `S1`.
`S1`	The *content* of object `S1`.
`siz(S1)`	Object `S1`’s size.
`siz(S1+S2)`	`S1`’s and `S2`’s combined size.

- **`strcat *S1,*S2`**

`strcat` catenates string `S2` to string `S1`. The operation may be characterized as `S1=S1+S2`. If `*S1` and `*S2` both point to the same string, the result will be `S1=S1+S1`. If `*S2` points to somewhere within `S1`, results are undefined.

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+5 → *S2
SP+1 → *S1
```

Upon return, carry will be set if any of the following is true:

```
siz(S1) > 32,767
siz(S2) > 32,767
siz(S1+S2) > 32,767
```

Assembly language synopsis:

```
PEA #S2_PTR >> 16    ;*S2 MSW
PEA #S2_PTR & $FFFF  ;*S2 LSW
PEA #S1_PTR >> 16    ;*S1 MSW
PEA #S1_PTR & $FFFF  ;*S1 LSW
JSR strcat
BCS ERROR
```

Exit register values:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    └┴┴┴┴┴┴────> entry value
```

- **`strchr *S,*C`**

`strchr` searches `S` for the first occurrence of the byte `C` and if found, returns a zero-based index to `C`’s position in `S`, as well as the number of instances of `C` in `S`. Note that `*C` points to a single byte in memory, not a character string.

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+5 → *C
SP+1 → *S
```

Upon return, carry will be set if the following is true:

```
siz(S) > 32,767
```

Assembly language synopsis:

```
PEA #C_PTR >> 16        ;*C MSW
PEA #C_PTR & $FFFF      ;*C LSW
PEA #S_PTR >> 16        ;*S MSW
PEA #S_PTR & $FFFF      ;*S LSW
JSR strchr
BCS ERROR
```

Exit register values:

```
.A: entry value
.B: entry value
.X: instances of C in S[1,2]
.Y: index to 1st C instance[1,2]
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ││││││││
    │││││││└───> 0: okay
    │││││││      1: error
    ││││└┴┴────> entry value
    │││└───────> 0[2]
    └┴┴────────> entry value
```

1) `$0000` if `C` was not found in `S`.
2) Entry value if function returns an error.

- **`strcmp *S1,*S2`**

`strcmp` compares string `S2` to string `S1`. Comparison is based upon the binary values of the strings' text field bytes, as well as the respective text field sizes. The results of comparison are returned in `SR` and as follows:

z	n	Meaning
1	X	S2 = S1
0	0	S2 > S1
0	1	S2 < S1

In the above table, `X` means "don't care." If `S1` and `S2` are both null, or if `*S1` and `*S2` point to the same string, the result will be `S2=S1`.

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+5 → *S2
SP+1 → *S1
```

Upon return, carry will be set if any of the following is true:

```
siz(S1) > 32,767
siz(S2) > 32,767
```

Assembly language synopsis:

```
PEA #S2_PTR >> 16        ;*S2 MSW
PEA #S2_PTR & $FFFF      ;*S2 LSW
PEA #S1_PTR >> 16        ;*S1 MSW
PEA #S1_PTR & $FFFF      ;*S1 LSW
JSR strcmp
BCS ERROR

BMI LESSER               ;S2 < S1
BEQ EQUAL                ;S2 = S1
BPL GREATER              ;S2 > S1
```

Exit register values:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    ||||||└────> 0: S2 != S1[1]
    ||||||       1: S2 = S1[1]
    |└┴┴┴┴─────> entry value
    └──────────> 0: S2 < S1[1]
                 1: S2 >= S1[1]
```

1) Entry value if function returns an error.

- **`strcpy *S1,*S2`**

`strcpy` copies `S2` to `S1`, overwriting `S1` with `S2`. If `*S1` and `*S2` point to the same string no operation will be performed and no error will occur. If `*S2` points to somewhere within `S1`, results are undefined.

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+5 → *S2
SP+1 → *S1
```

Upon return, carry will be set if the following is true:

```
siz(S2) > 32,767
```

Assembly language synopsis:

```
PEA #S2_PTR >> 16       ;*S2 MSW
PEA #S2_PTR & $FFFF     ;*S2 LSW
PEA #S1_PTR >> 16       ;*S1 MSW
PEA #S1_PTR & $FFFF     ;*S1 LSW
JSR strcpy
BCS ERROR
```

Exit register values:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    └┴┴┴┴┴┴────> entry value
```

- **`strdel *S,*I,*N`**

`strdel` deletes a substring from string `S`. The beginning of the substring to be deleted is defined by index `I`. The size of the substring to be deleted is defined by `N`.

- ○ `I` is an integer and may be any quantity. However, if `I>=siz(S)`, no operation will be performed and no error will occur.
- ○ `N` is an integer and may be any quantity. However, if `N=0`, no operation will be performed and no error will occur.
- ○ If `siz(S)=0`, no operation will be performed and no error will occur.
- ○ If `(I+N)<siz(S)` is true, `N` bytes will be logically removed from `S` starting at byte `I`, and the remainder of the string will be shifted to "close the gap." Memory previously occupied by the remainder of the string will contain "garbage."
- ○ If `(I+N)>=siz(S)` is true, `S` will be truncated to `I` bytes. Memory previously occupied by the remainder of the string will contain "garbage."

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+9 → *N
SP+5 → *I
SP+1 → *S
```

Upon return, carry will be set if the following is true:

```
siz(S) > 32,767
```

Assembly language synopsis:

```
PEA #N_PTR >> 16        ;*N MSW
PEA #N_PTR & $FFFF      ;*N LSW
PEA #I_PTR >> 16        ;*I MSW
PEA #I_PTR & $FFFF      ;*I LSW
PEA #S_PTR >> 16        ;*S MSW
PEA #S_PTR & $FFFF      ;*S LSW
JSR strdel
BCS ERROR
```

Exit register values:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    └┴┴┴┴┴┴────> entry value
```

- **`strins *S1,*S2,*I`**

`strins` inserts string `S2` into string `S1`. The point of insertion is defined by zero-based index `I`. `S1` will expand in size by the size of `S2`. If `*S2` points to somewhere within `S1`, results are undefined.

- ○ `I` is an integer and may be any quantity. However, if `I>=siz(S1)` is true, no operation will be performed and no error will occur.
- ○ If `siz(S1)=0`, no operation will be performed and no error will occur. In such a case, the `strcpy` function should be used to copy `S2` to `S1`.
- ○ If `siz(S2)=0`, no operation will be performed and no error will occur.
- ○ If `*S1` and `*S2` point to the same string, no operation will be performed and no error will occur.

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+9 → *I
SP+5 → *S2
SP+1 → *S1
```

Upon return, carry will be set if any of the following is true:

```
siz(S1) > 32,767
siz(S2) > 32,767
siz(S1+S2) > 32,767
```

Assembly language synopsis:

```
PEA #I_PTR >> 16        ;*I MSW
PEA #I_PTR & $FFFF      ;*I LSW
PEA #S2_PTR >> 16       ;*S2 MSW
PEA #S2_PTR & $FFFF     ;*S2 LSW
PEA #S1_PTR >> 16       ;*S1 MSW
PEA #S1_PTR & $FFFF     ;*S1 LSW
JSR strins
BCS ERROR
```

Exit register values:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    └┴┴┴┴┴┴────> entry value
```

- **`strlen *S`**

`strlen` reports `siz(S).`

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+1 → *S
```

Upon return, carry will be set if the following is true:

```
siz(S) > 32,767
```

Assembly language synopsis:

```
PEA #S_PTR >> 16        ;*S MSW
PEA #S_PTR & $FFFF      ;*S LSW
JSR strlen
BCS ERROR
```

Exit register values:

```
.C: siz(S)[1]
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    ||||||└────> 0: siz(S) > 0[1]
    ||||||       1: siz(S) = 0[1]
    |||└┴┴─────> entry value
    ||└────────> 0[1]
    └┴─────────> entry value
```

1) Entry value if function returns an error.

- **`strpad *S1,*S2,*L,J,FB`**

  `strpad` copies string `S2` to `S1` and then modifies `S1` according to `L`, `J` and `FB`.

  - `*L` points to an integer that determines the final size of `S1`. `L`'s effect is as follows:

    `L<siz(S2)`: `S1` will be a truncated copy of `S2`.

    `L=siz(S2)`: `S1` will be an exact copy of `S2`.

    `L>siz(S2)`: `S1` will be a padded copy of `S2`. The value passed as `FB` will be used as the padding byte (see below).

    The permissible range for `L` is `0` to `32,767` inclusive. If `L=0` then `S1` will be a null string.

  - `J` is a value that specifies how `S1` is to be justified if padding is required to achieve size `L`. `J` will have no effect if `siz(S2)>=L` is true. Behavior is as follows:

    `J=0`: `S1` will be justified left. For example, if `S2="ABC"`, `L=7`, `J=0` and `FB=$2A` (`FB` is described below) then `S1` will be `"ABC****"`. If `stringmacs.asm` has been assembled in your program, the symbol `_JUSTLFT` may be used to represent `J`.

    `J=1`: `S1` will be justified right. For example, if `S2="ABC"`, `L=7`, `J=1` and `FB=$2A` then `S1` will be `"****ABC"`. If `stringmacs.asm` has been assembled in your program, the symbol `_JUSTRIT` may be used to represent `J`.

    `J=2`: `S1` will be centered. For example, if `S2="ABC"`, `L=7`, `J=2` and `FB=$2A` then `S1` will be `"**ABC**"`. If `stringmacs.asm` has been assembled in your program, the symbol `_JUSTCTR` may be used to represent `J`.

    Any other `J` value will be treated as `J=0`. `J` will be ignored if `siz(S2)>=L` is true.

  - `FB` is a value that will be used as the fill byte if padding is applied to `S1`. If `FB` is a null (`$00`) an ASCII blank (`$20`) will be substituted. Caution should be exercised in using an `FB` value that is outside of the printable ASCII range if `S1` is to be displayed on a screen or printer. `FB` will be ignored if `siz(S2)>=L` is true.

  - If `*S1` and `*S2` point to the same string, no operation will be performed and no error will occur. If `*S2` points to somewhere within `S1`, results are undefined.

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+15 → FB
SP+13 → J
SP+9  → *L
SP+5  → *S2
SP+1  → *S1
```

Upon return, carry will be set if any of the following is true:

```
siz(S2) > 32,767
L > 32,767
```

Assembly language synopsis:

```
PEA #FB                 ;padding (fill) byte
PEA #J                  ;_JUSTLFT | _JUSTRIT | _JUSTCTR
PEA #I_PTR >> 16        ;*L MSW
PEA #I_PTR & $FFFF      ;*L LSW
PEA #S2_PTR >> 16       ;*S2 MSW
PEA #S2_PTR & $FFFF     ;*S2 LSW
PEA #S1_PTR >> 16       ;*S1 MSW
PEA #S1_PTR & $FFFF     ;*S1 LSW
JSR strpad
BCS ERROR
```

Exit register values:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    └┴┴┴┴┴┴────> entry value
```

- **`strpat *S,*P`**

`strpat` compares string `P`, a “pattern,” to string `S` and returns the Boolean results `LIKE` or `UNLIKE`. A return of `LIKE` means the pattern was found in `S`. The presence of the metacharacters **?** (character wildcard) and ***** (string wildcard) in `P` will affect the outcome of the comparison. Pattern matching behaves as follows:

? When present in P, a character wildcard will match exactly one character in S. For example, `fo?bar` will match `foobar`, `fo0bar`, `forbar`, etc. `????` will match `S` if `S` is exactly four bytes in size, with the bytes in `S` being anything, including `????`.

* When present in P, a string wildcard will match a sequence of bytes in S. For example, `ab*` will match anything that begins with `ab`, such as `abcde` or `abracadabra`. `*yz` will match anything that ends with `yz`, such as `abcxyz`. `*lm*` will produce a match if `S`’s text field contains `lm`. `ab*lm*yz` will produce a match if `S`’s text field begins with `ab`, has `lm` anywhere in the middle, and ends with `yz`.

Combinations of wildcards in `P` may be used in various ways. For example, `?*` will match `S` if it is non-null. Similarly, `???*` will match if `S` is three or more bytes in size, as would `*???`. The somewhat strange pattern `???*???` would match `S` if the latter contains at least six bytes.

The special case of `*` being the only byte in `P` will match with anything, even a null string. If `P` consists solely of multiple `*` bytes, such as `***`, `strpat` will treat them as a single instance of `*`. Similarly, `ab***yz` will be logically reduced to `ab*yz`. If `P` has no wildcards, `strpat` will act in a similar fashion to the `strcmp` function, but with slower execution and without the “lesser-than” and “greater-than” results of `strcmp`.

The comparison result is indicated by the microprocessor’s `z` flag as follows:

z	Meaning
1	P **LIKE** S
0	P **UNLIKE** S

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+5 → *P
SP+1 → *S
```

Upon return, carry will be set if any of the following is true:

```
siz(S) > 32,767
siz(P) > 32,767
```

Assembly language synopsis:

```
PEA #P_PTR >> 16          ;*P MSW
PEA #P_PTR & $FFFF        ;*P LSW
PEA #S_PTR >> 16          ;*S MSW
PEA #S_PTR & $FFFF        ;*S LSW
JSR strpat
BCS ERROR

BEQ LIKE                  ;S is LIKE P
BNE UNLIKE                ;S is UNLIKE P
```

Exit register values:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    ||||||└────> 0: S UNLIKE P[1]
    ||||||       1: S LIKE P[1]
    └┴┴┴┴┴─────> entry value
```

1) Entry value if function returns an error.

- **`strstr *S1,*S2`**

  `strstr` searches string `S1` for string `S2` and if found, returns both a zero-based index and a pointer to where `S2` was encountered in `S1`. The `z` bit in `SR` is used to indicate if `S2` was found in `S1`—refer to the assembly language synopsis for details. “Not found” status will be returned if either string is null. If `*S2` points to anywhere within `S1`, a “found” status will be returned.

  Prior to calling this function, the parameter stack frame must be as follows:

  ```
  SP+5 → *S2
  SP+1 → *S1
  ```

  Upon return, carry will be set if any of the following is true:

  ```
  siz(S1) > 32,767
  siz(S2) > 32,767
  ```

  Assembly language synopsis:

  ```
  PEA #S2_PTR >> 16        ;*S2 MSW
  PEA #S2_PTR & $FFFF      ;*S2 LSW
  PEA #S1_PTR >> 16        ;*S1 MSW
  PEA #S1_PTR & $FFFF      ;*S1 LSW
  JSR strstr
  BCS ERROR

  BEQ FOUND                ;S2 found in S1
  BNE NOTFOUND             ;S2 not found in S1
  ```

  Exit register values (next page):

```
.C: index to S2's location in S1[1,2]
.X: pointer to S2's location in S1 LSW[1,2]
.Y: pointer to S2's location in S1 MSW[1,2]
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    ||||||└────> 0: S2 not found in S1[1]
    ||||||       1: S2 found in S1[1]
    ||||└┴─────> entry value
    ||└┴───────> 0[1]
    └┴─────────> entry value
```

1) Entry value if function returns an error.
2) $0000 if S2 was not found in S1.

- **`strsub *S1,*S2,*I,*N`**

  `strsub` copies a substring from `S2` to `S1`, overwriting `S1`. If `*S2` and `*S1` point to the same string, no operation will be performed and no error will occur. If `*S2` points to somewhere within `S1`, results are undefined.

  - `I` is the starting index into `S2`. If `I>=siz(S2)`is true, no operation will be performed and no error will occur.
  - `N` is the maximum number of bytes to be copied from S2. If `N=0`, no operation will be performed and no error will occur.
  - If `(I+N)<siz(S2)` is true, `N` bytes will be copied from `S2` starting at byte `I` in `S2`.
  - If `N>0` and `(I+N)>=siz(S)` is true, `siz(S2)-I` bytes will be copied from `S2` starting at byte `I`.
  - If `siz(S2)=0`, no operation will be performed and no error will occur.

  Prior to calling this function, the parameter stack frame must be as follows:

```
sp+13 → *N
SP+9  → *I
SP+5  → *S2
SP+1  → *S1
```

  Upon return, carry will be set if the following is true:

```
siz(S2) > 32,767
```

  Assembly language synopsis:

```
PEA #N_PTR >> 16        ;*N MSW
PEA #N_PTR & $FFFF      ;*N LSW
PEA #I_PTR >> 16        ;*I MSW
PEA #I_PTR & $FFFF      ;*I LSW
PEA #S2_PTR >> 16       ;*S2 MSW
PEA #S2_PTR & $FFFF     ;*S2 LSW
PEA #S1_PTR >> 16       ;*S1 MSW
PEA #S1_PTR & $FFFF     ;*S1 LSW
JSR strsub
BCS ERROR
```

Exit register values:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ||||||||
    |||||||└───> 0: okay
    |||||||      1: error
    └┴┴┴┴┴┴────> entry value
```

- **strsws *S,OP**

`strsws` strips “white space” from string `S`, according to the operation defined by `OP`. “White space” refers to leading and/or trailing blanks (ASCII 32) and/or horizontal tabs (ASCII 9) that are part of `S`, e.g., `"    ABCDE   "`.

○ `OP`, which is a bit-wise value, defines the operation to be performed:

```
xx000000
||
|└─────────> 1: strip leading whitespace
└──────────> 1: strip trailing whitespace
```

Both operations may be performed in a single call.

○ If `siz(S)=0`, no operation will be performed and no error will occur.

Prior to calling this function, the parameter stack frame must be as follows:

```
SP+5  → OP
SP+1  → *S1
```

Upon return, carry will be set if the following is true:

```
siz(S) > 32,767
```

Assembly language synopsis:

```
PEA #OP                 ;OP
PEA #S_PTR >> 16        ;*S MSW
PEA #S_PTR & $FFFF      ;*S LSW
JSR strsws
BCS ERROR
```

Exit register values are on the next page:

```
.A: entry value
.B: entry value
.X: entry value
.Y: entry value
DB: entry value
DP: entry value
PB: entry value
SR: nvmxdizc
    ││││││││
    │││││││└───> 0: okay
    │││││││      1: error
    └┴┴┴┴┴┴────> entry value
```

DISCLAIMER